# AN1053: *Bluetooth*® Device Firmware Update over UART for EFR32xG1 and BGM11x Series Products

This application note describes the legacy UART DFU (Device Firmware Update) mechanism used in the Silicon Labs Bluetooth SDK (Software Development Kit) for EFR32xG1 SoCs and BGM11x and BGM121/BGM123 modules. UART DFU enables deployment of firmware updates to devices in the field, making it possible to introduce new features or other changes after a product has been launched.

The Silicon Labs Bluetooth SDK allows designers to easily add UART DFU capability in their products. The UART DFU functionality is handled completely by the bootloader and does not require any extra code in the user application. This documents explains how to configure a C-based project to support UART DFU with the legacy bootloader and how to test firmware updates with the host example that is provided in the SDK.

**KEY POINTS**

- UART DFU basic procedure
- Enabling DFU in your projects
- Creating update images
- UART DFU host example

# 1. Introduction

In Bluetooth SDK v 2.3.0.0, Silicon Labs introduced a new Gecko Bootloader, which is required for the new EFR32xG12 platform and all future parts. It adds many improvements and features, including a configurable base framework, security options, support for multiple update images, and in-field upgrades for the bootloader itself. While it is required for the new EFR32xG12 parts, the Gecko Bootloader can also be used on some earlier parts. Section 1.2 Bootloaders and Supported Parts details which bootloaders can be used with which parts.

This document describes how to use the legacy UART DFU (Device Firmware Update) with the Silicon Labs Bluetooth SDK. For more information on the new Gecko Bootloader, see *UG266: Silicon Labs Gecko Bootloader User's Guide*. For background on bootloader categories and the various Silicon Labs bootloaders, see *UG103.6: Application Development Fundamentals: Bootloading*.

The following topics are covered in this document:

- How the basic UART DFU procedure works.
- The basic configurations required to enable UART DFU in a project.
- Additional configurations that may be needed to enable UART DFU on custom hardware.
- Creation of update images.
- Testing UART DFU with the host example code provided in the SDK.

## 1.1  Prerequisites

The UART DFU solution discussed in this document applies to Silicon Labs Bluetooth SDK versions 2.0.0 and later. It is not possible to update from older stack versions (1.0.x) to 2.0 or later using UART DFU.

You should be familiar with developing NCP (Network Co-Processor) applications using the Silicon Labs Bluetooth SDK. The basics of NCP mode application development is discussed in *AN1042: Using the Silicon Labs Bluetooth® Stack in Network Co-Processor Mode*.

For testing the UART DFU, using one of the Silicon Labs Bluetooth-enabled Wireless Starter Kits (WSTK) is recommended, but not mandatory.

## 1.2  Bootloaders and Supported Parts

If you are designing for the EFR32xG12 or later parts, you must use the Gecko Bootloader. For some other parts, you have the option of using the Gecko Bootloader if you want to take advantage of its new functions. The following table shows bootloader support by part category.

**Table 1.1.  Bootloader Part Support**

|  | Legacy UART Bootloader | Gecko Bootloader |
|---|---|---|
| EFR32xG1 256kB Flash parts | Yes | Yes |
| EFR32xG1 128kB Flash parts | Yes | Limited |
| BGM11x | Yes | Yes |
| BGM121 / BGM123 | Yes | Yes |
| EFR32xG12 | No | Yes |
| Future releases SoC and Module | No | Yes |

Although the Gecko Bootloader supports field-upgradable bootloader configuration, the Bluetooth legacy bootloaders are not field-upgradable. If a legacy bootloader needs to be updated, the image must be flashed using the SWD (Serial Wire Debug).

## 1.3  UART DFU Basics

This section explains some UART DFU basics. More detailed instructions specific to application flow are included later in this document.

### 1.3.1 Bootloader

The UART DFU is implemented almost entirely in the bootloader. Starting with Silicon Labs Bluetooth SDK version 2.0, the bootloader occupies a fixed 16-kB area in the beginning of flash memory. The SDK includes a default bootloader that supports UART DFU. The bootloader is also provided in source code format. Typically, application developers do not need to modify the bootloader code, but can start with the source code if customization is required.

Using UART DFU, it is possible to perform a full firmware update (including the Silicon Labs Bluetooth stack, GATT database, hardware configurations, and application code). However, the bootloader itself cannot be updated using UART DFU.

### 1.3.2 UART Settings

The default UART configuration in the bootloader is as follows:

- Baud rate 115200
- UART TX: pin PA0
- UART RX: pin PA1
- No hardware flow control

This configuration works with most of the Silicon Labs Bluetooth-enabled WSTKs. The pin mappings and other UART settings can be changed. However, the easiest way to get started is to use the default configuration listed above.

Hardware flow control is optional. By default, flow control is not enabled.

### 1.3.3 Update Image Format

The file format of the update images is .EBL, a proprietary Silicon Labs file format used by several radio protocol stacks as well as the Silicon Labs Bluetooth stack. EBL files include CRC checksums and other metadata that can be used to validate the integrity of the update image.

EBL images are generated during the user application build process. Note that typically three *.ebl files are generated. For example, when building the NCP Target – Empty example project from the Bluetooth SDK, after running the EBL creation script (**create_ebl_files.bat /.sh**) from the workspace directory, three update images are created to the output_ebl directory, as shown in the following figure.

| Name | Date modified | Type | Size |
|---|---|---|---|
| app.ebl | 5/16/2017 3:51 PM | EBL File | 25 KB |
| full.ebl | 5/16/2017 3:51 PM | EBL File | 141 KB |
| stack.ebl | 5/16/2017 3:51 PM | EBL File | 117 KB |

**Figure 1.1. EBL Files Generated for the NCP Target - Empty Example**

Only the file ending **full.ebl** is intended for UART DFU updates. The other two EBL files are used for Over-The-Air (OTA) updates. They should be ignored in projects that use UART DFU.

The UART DFU update file contains the entire firmware image, excluding the bootloader. Image files (.bin and .s37) found from the workspace are to be used when programming the target device using a flash programmer. These files include the bootloader, and therefore is 16 kB larger than the UART DFU update image.

User host applications do not need to understand the internal structure of the EBL file. The update image includes a CRC checksum that is automatically generated as part of the firmware build process. This CRC checksum is used by the bootloader to verify the integrity of the update image. To perform UART DFU, the NCP host application only needs to know the exact size of the EBL file.

### 1.4 UART DFU Process

The basic steps involved in the UART DFU are as follows:

1. Boot the target device into DFU mode (by sending `dfu_reset(1)` ).
2. Wait for the `DFU boot` event.
3. Send the command `Flash Set Address` to start the firmware update
4. Send the entire contents of the EBL update image (using the command `DFU flash upload`).
5. After sending all data, the host sends the command `DFU flash upload finish`.
6. To finalize the update, the host resets the target device into normal mode (by sending `dfu_reset(0)` ).

A detailed description of the DFU-related BGAPI commands is found in the *Bluetooth Smart Software API Reference Manual*.

At the beginning of the update, the NCP host uses the command `Flash Set Address` to define the start address. The start address should be always set as zero. During the data upload (step 4 above) the target device calculates the flash offset automatically. The host does not need to explicitly set any write offset.

The SDK includes a minimal host example program that performs UART DFU following the steps listed above. Chapter 3. UART DFU Host Example describes in more detail how to build and run this example.

## 2. UART DFU in C-Based NCP Applications

### 2.1 Bootloader Configuration in C Projects

All the C-based NCP examples in the Silicon Labs Bluetooth SDK use a fixed bootloader image by default. The default bootloader is configured to support OTA updates. Therefore, any C-based NCP project that needs to support UART DFU has to be configured to use a UART DFU capable bootloader instead of the default one.

The default bootloader (**binbootloader.o** file) that is used for C applications is located in following directories:

**SDK version 2.0.x.x – 2.1.x.x**

C:\SiliconLabs\SimplicityStudio\v4\developer\stacks\ble\v2.x.x.x\protocol\bluetooth_2.0\lib\

**Gecko Suite v1.0, Bluetooth v2.3.x.x onwards**

C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.x\protocol\bluetooth_2.x\lib

The pre-compiled **binbootloader.o** is linked to the example projects in the IAR linker settings. The following screenshot is taken from the **ncp-empty-target** example in SDK 2.0.0.
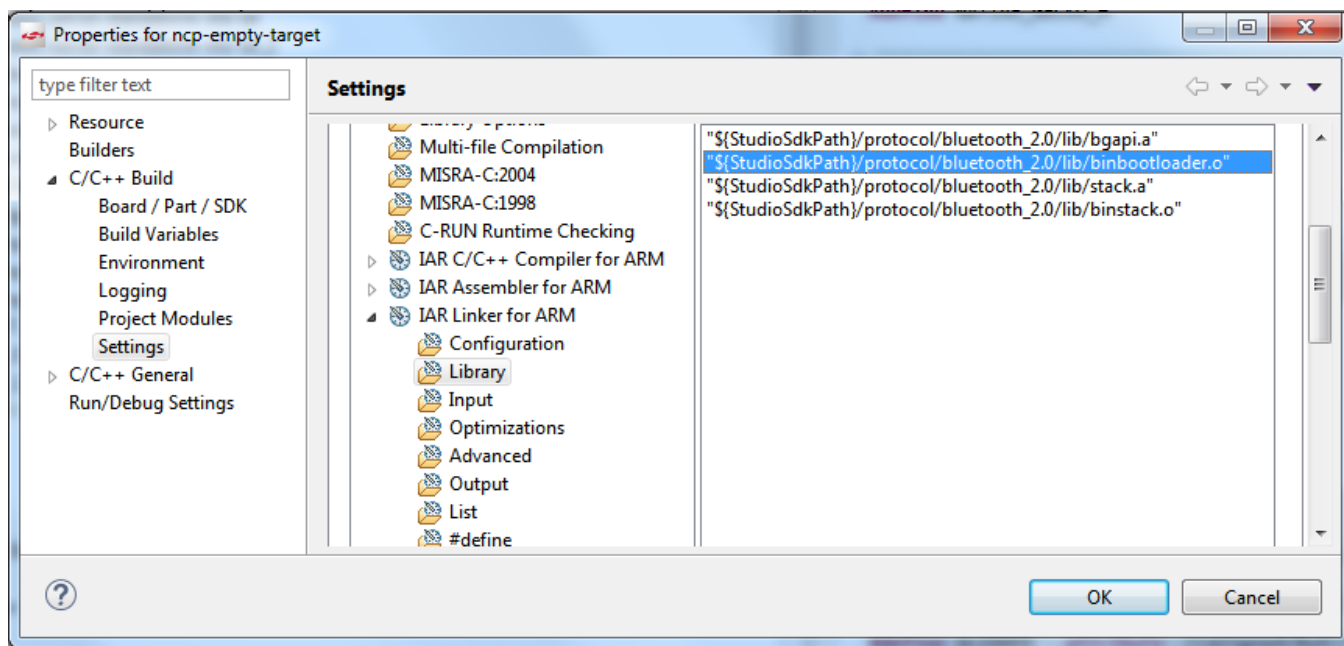


**Figure 2.1. Default Bootloader in "ncp-empty-target" Example (no UART DFU Support)**

## 2.2 Preparing a UART DFU-Capable Bootloader

The bootloader source code is included in the SDK in following directory:

**SDK version 2.0.x.x – 2.1.x.x**

C:\SiliconLabs\SimplicityStudio\v4\developer\stacks\ble\v2.x.x.x\protocol\bluetooth_2.x\ble_stack\bootloader

**Gecko Suite v1.x, Bluetooth v2.x.x.x onward**

C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.x\protocol\bluetooth_2.x\ble_stack\bootloader

The **bootloader** directory has a subfolder **iar** that includes an IAR project file (bootloader.ewp). Building the bootloader requires the IAR toolchain and a valid IAR license.

**Note:** It is recommended to make a copy of the bootloader project before changing any build settings or modifying the source files, instead of modifying the original source code directly. The bootloader project can be duplicated by simply copying the whole bootloader directory and renaming it, for example, **bootloader_dfu**.

After creating a copy of the bootloader project it is good to first check that the project builds without any errors.

The same bootloader source is used for both the OTA and UART DFU versions of the bootloader. To enable the UART DFU version, the preprocessor directive `UART_DFU` must be defined. This is defined automatically in the IAR project settings. The setting can be verified from the C/C++ Compiler settings as shown in the following figure.
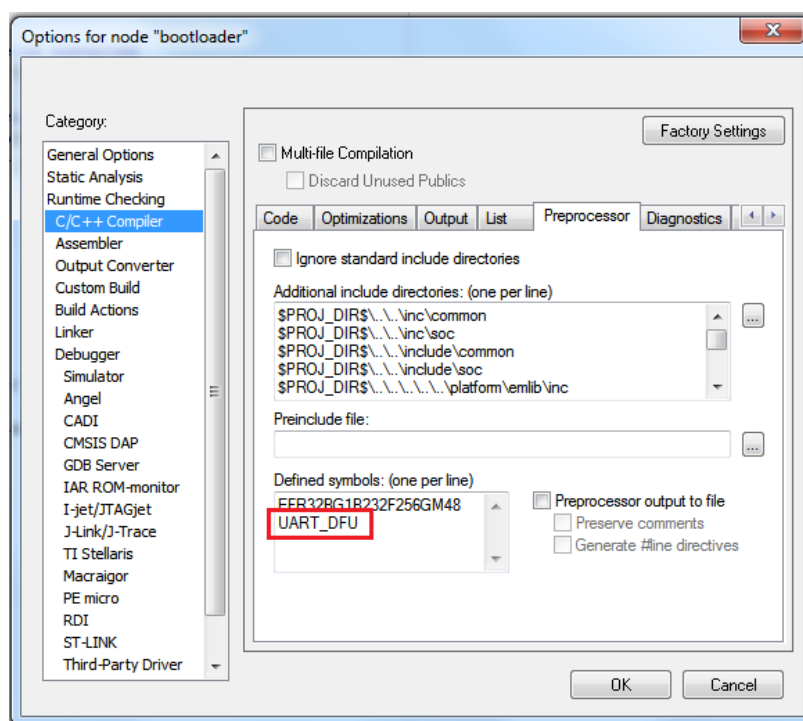


**Figure 2.2. UART_DFU Precompiler Directive in Bootloader Project Settings**

Typically it is enough to make sure that the bootloader UART configuration matches the UART settings used in the NCP project. UART settings for the bootloader are collected in a header file named **uart_config.h**. The basic UART configurations that may need to be adjusted to match the NCP project settings are highlighted in the following figure.

```
main.c   uart_config.h

 1   #ifndef UART_CONFIG_H
 2   #define UART_CONFIG_H
 3   #include <stddef.h>
 4   #include "em_usart.h"
 5
 6   #define WSTK_VCOM 0
 7
 8   static const char *tx_pin = "PA0";
 9   static const char *rx_pin = "PA1";
10   static const char *cts_pin = NULL;
11   static const char *rts_pin = NULL;
12
13   static const USART_InitAsync_TypeDef uartInit =
14   {
15       .enable       = usartDisable,    // Wait to enable the transmitter and receiver
16       .refFreq      = 38000000,        // 38MHz HFRCO
17       .baudrate     = 115200,          // Desired baud rate
18       .databits     = usartDatabits8,  // 8 data bits
19       .mvdis        = false,           // Use majority voting
20       .prsRxEnable  = false,           // Not using PRS input
21       .prsRxCh      = usartPrsRxCh0,   // Doesn't matter which channel we select
22       .oversampling = usartOVS16,      // Set oversampling value to x16
23       .parity       = usartNoParity,   // No parity
24       .stopbits     = usartStopbits1,  // 1 stop bit
25   };
26
27
28   #endif
29
```

**Figure 2.3. Bootloader UART Settings in uart_config.h**

The `WSTK_VCOM` directive can be used to enable the additional GPIO settings that are necessary in projects running on a WSTK and using the on-board USB-to-UART converter as the host UART.

By default, `WSTK_VCOM` is defined as zero, meaning that the project is not built for WSTK.

The UART pin mappings are defined as string constants on lines 8..11. If flow control is not desired then the CTS and RTS pin names are set as NULL.

The UART baud rate can be changed by editing the `.baudrate` member of the `uartInit` struct (line 17).

**Note:** When using a WSTK and VCOM as the UART host interface, it is recommended to use the default baud rate 115200.

After checking that the UART configurations are correct, the project needs to be built. When testing with a WSTK it is enough to modify the definition of WSTK_VCOM to a non-zero value. No other changes are needed.

As a result of a successful build a file named **binbootloader.o** is generated in the IAR output directory. This is the customized bootloader that you need to link with your NCP project, as described in the next section.

The bootloader project includes a post-build step that uses the GNU assembler (arm-none-eabi-as) for generating the **binbootloader.o** output. This is defined in the project settings (Build Actions -> Post-build command line). The default post-build action is configured as:

```
arm-none-eabi-as -mthumb -mcpu=cortex-m4 -march=armv7e-m $PROJ_DIR$\..\binbootloader.S -I $EXE_DIR$
-o $EXE_DIR$\binbootloader.o
```

The post-build step may fail if you do not have arm-none-eabi-as in your system path. In this case, you can fix the build by specifying the absolute path to arm-none-eabi-as and editing the post-build command line setting as follows:

```
C:\SiliconLabs\SimplicityStudio\v4\developer\toolchains\gnu_arm\4.9_2015q3\bin\arm-none-eabi-as
-mthumb -mcpu=cortex-m4 -march=armv7e-m $PROJ_DIR$\..\binbootloader.S -I $EXE_DIR$ -o $EXE_DIR$\binbootloader.o
```

## 2.3 Replacing the Default Bootloader

After you have created a customized **binbootloader.o** image that is configured to match your NCP project settings, the bootloader must be linked into your project so that it replaces the default bootloader that comes with the SDK. The following figure shows an example of modified linker settings, where the default **binbootloader.o** has been removed from the list and replaced with a custom **binbootloader.o** that is copied to the project workspace.
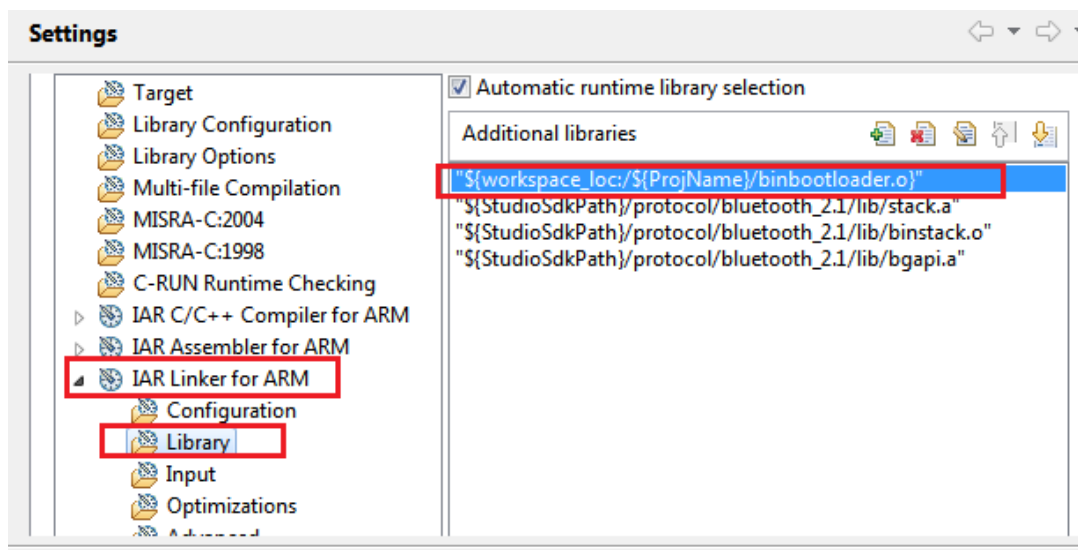


**Figure 2.4. Custom Bootloader Added to Project Linker Settings**

After replacing the default bootloader with your customized version, build your NCP project and flash it to the NCP target. A simple way to check if the bootloader change was successful is to observe the events generated by the NCP target after reset. The UART_DFU bootloader generates the BGAPI event `dfu_boot` after reset. This event is not generated if the target device is linked with the OTA bootloader, or if the UART configurations are not set correctly. The purpose of the automatically generated `dfu_boot` event is explained later in section 4.2 DFU Window at Boot.

## 2.4 Generating EBL Update Images

Starting from SDK version 2.1.0, the example projects include a script in the project root folder for generating the update images. There are two scripts, named:

- **create_ebl_files.bat** (for Windows)
- **create_ebl_files.sh** (for Linux / Mac)

Running the **create_ebl_files** script creates three EBL files in a subfolder named **output_ebl**. The file named **full.ebl** is the update image used for UART DFU. The other two files (**app.ebl**, **stack.ebl**) are related to OTA updates and they can be ignored.
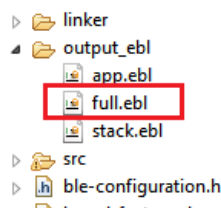


**Figure 2.5. Files Generated by the Create_ebl_files Script**

## 3. UART DFU Host Example

This chapter explains how to use the UART DFU host example for testing UART DFU.

Before testing UART DFU, you need to have:
• An NCP target device that has been configured to support UART DFU (firmware built with SDK 2.0.0 or later)
• The EBL file (full update) generated from your NCP project

The UART DFU host example is a C program that is located under the SDK examples in following directory:

C:\SiliconLabs\SimplicityStudio\v4\developer\stacks\ble\v2.x.x.x\app\bluetooth_2.x\examples_ncp_host\uart_dfu

In Windows this program can be built using, for example, MinGW or Cygwin. In Linux or Mac the program can be built using the GCC toolchain.

The project is built by running make (or mingw32-make) in the project root directory. After a successful build, an executable named **uart-dfu.exe** is created in subfolder **exe**.

Before running the example you need to check the COM port number associated with your NCP target. For more details, refer to *AN1042: Using the Silicon Labs Bluetooth® Stack in Network Co-Processor Mode*.

The uart-dfu.exe program requires three command line arguments:
• COM port number
• Baud rate
• Name of the (full) EBL file

Example usage and expected output:

```
./uart-dfu.exe COM42 115200 full.ebl
Syncing..DFU OK
Bootloader version: 4 (0x4)
........
........
finish
```

Using a WSTK with the default 115200 baud rate, the whole update procedure takes about 40 seconds to complete.

The number of bytes uploaded in one DFU flash upload command is configurable. The UART DFU host example included in the SDK uses a 48-byte payload. The maximum usable payload length in Silicon Labs Bluetooth SDK 2.1.0 is 128 bytes. The maximum number of bytes sent in one command is specified using a C preprocessor directive named MAX_DFU_PACKET. The value of MAX_DFU_PACKET must be divisible by four.

# 4. Error Handling in UART DFU

## 4.1 Corrupted or Incomplete Image

The UART DFU procedure may fail if the update image is either corrupted or if data upload is interrupted for some reason. Both of these conditions are detected by the CRC check that is performed by the UART DFU bootloader before jumping into the main program. If the CRC check fails then the bootloader does not jump into the main program but stays in bootloader mode, allowing the UART DFU procedure to be repeated.

The code that performs CRC check (from bootloader **main.c**) is shown below.

```
226    //check image crc
227    //skip crc check if original software is installed
228    if ((saat->imageCrc != IMAGE_CRC_MAGIC) || (saat->timeStamp != IMAGE_TIMESTAMP_MAGIC))
229    {
230        /* if crc error send event and erase first memory page */
231        if (!check_image_crc((uint8_t *)BOOTLOADER_SIZE)) {
232            gecko_evt_dfu_boot_failure(bg_err_security_image_checksum_error);
233            do
234            {
235                uart_input();
236            } while(1);
237        }
238    }
239
```

**Figure 4.1. CRC Check in Bootloader**

Note that the CRC check is skipped if the firmware has been programmed directly via the debug interface. This makes it possible to launch and debug a C-based NCP application directly from Simplicity Studio or the IAR IDE, without having to create a full EBL image each time the program is built.

If the CRC check fails then the bootloader generates event `evt_dfu_boot_failure` with the reason code set as `0x0B03` (image_checksum_error). The execution then stays in bootloader mode, so that the NCP host can program a valid EBL image.

## 4.2 DFU Window at Boot

The `UART DFU` procedure is normally initiated so that the NCP host reboots the NCP target device into DFU mode using the BGAPI command `cmd_dfu_reset(1)`. In some cases, it may be impossible to trigger DFU reboot this way. For example, if the NCP target has been programmed with an invalid UART configuration, then the main application cannot receive any BGAPI commands.

The default UART bootloader in the Silicon Labs Bluetooth SDK includes a boot delay that allows the NCP host to start DFU update immediately after reset, before the main application has even started. The first event generated after reset is always `evt_dfu_boot`, indicating that the bootloader is ready to accept UART DFU commands. The bootloader then waits approximately one second. If the host does not start UART DFU during this delay then the bootloader continues execution into the main program, which results in the normal system boot event (`evt_system_boot`).

The boot delay is implemented in file **main.c**, function `main_loop()`, as shown in the following figure

```
130    void main_loop()
131    {
132        volatile uint32_t l=1000000;
133        do
134        {
135            uart_input();
136    #if !DEBUG_STAY
137            if(!dfu_active&&(*MAIN_APP!=0xFFFFFFFF))
138                l--;
139    #endif
140        }while(l);
141    }
142
```

**Figure 4.2. DFU Boot Delay**

As shown in the above figure, the loop variable `l` is decremented if the `dfu_active` variable is not set. This causes the while-loop to terminate when the counter reaches zero. If the NCP host sends command `Flash Set Address` during the DFU window then the `dfu_active` variable is set and the countdown is stopped.

The DFU boot window allows the NCP target device to be reprogrammed using UART DFU even if the main application is completely missing.
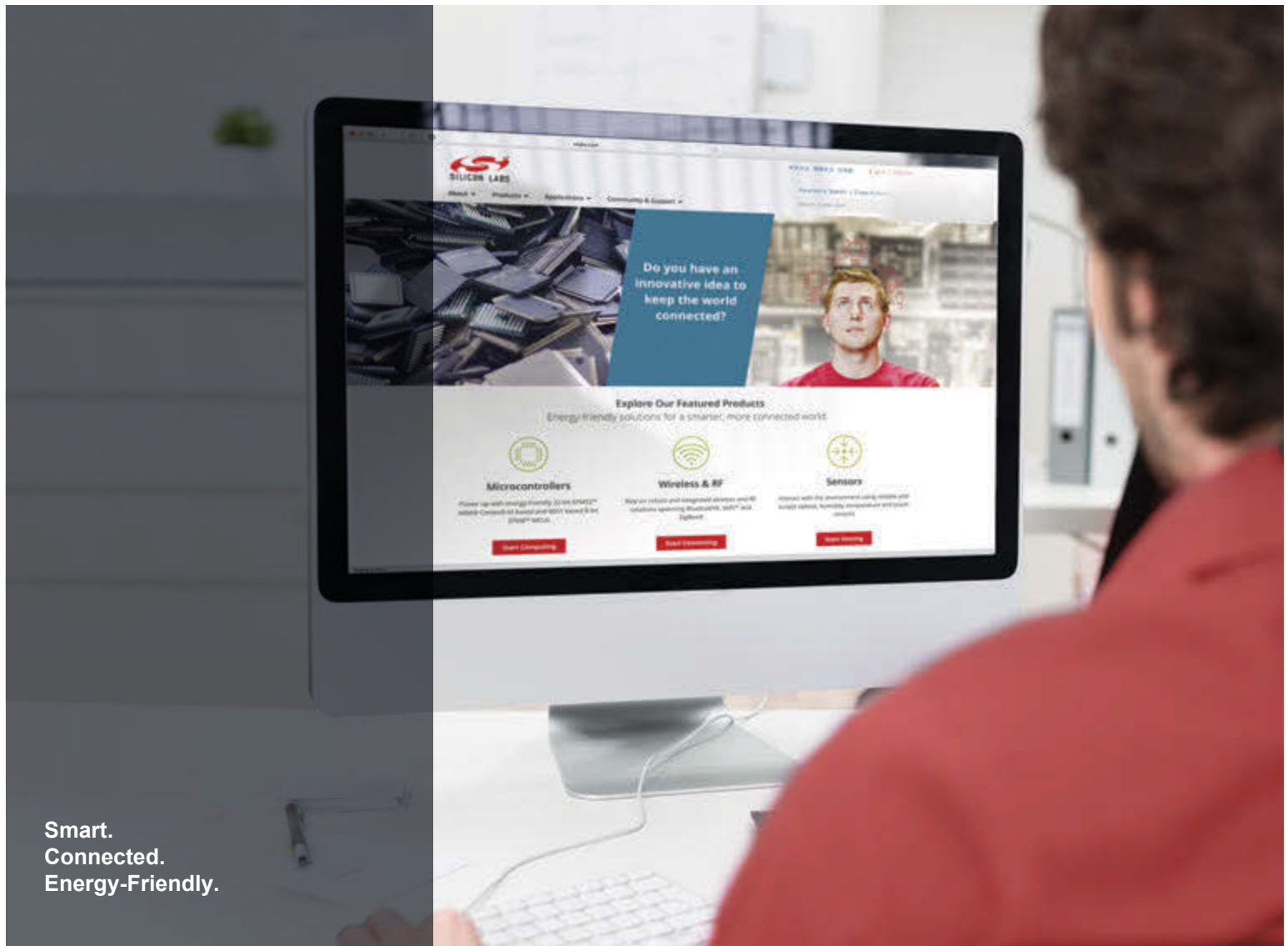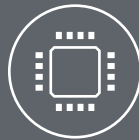
Smart.
Connected.
Energy-Friendly.

| Products | Quality | Support and Community |
|----------|---------|----------------------|
| www.silabs.com/products | www.silabs.com/quality | community.silabs.com |

**Silicon Laboratories Inc.**
**400 West Cesar Chavez**
**Austin, TX 78701**
**USA**

**http://www.silabs.com**